4.Spring

什么是Spring?

Spring | Home

Spring是一个基于Java的开源框架，它最初由Rod Johnson在2002年发布，并在随后的几年中逐渐发展完善。Spring框架的主要目的是简化企业级Java应用程序的开发过程，通过提供一系列的功能和工具，帮助开发者更高效地构建和维护应用程序。以下是对Spring框架的详细解读：

1. 核心特点

- 轻量级：Spring框架在设计时注重轻量级，其完整的框架可以发布在一个大小只有几兆的JAR文件中，并且所需的处理开销也非常小。
- 控制反转（IoC）：Spring框架的核心思想之一是控制反转，也称为依赖注入（DI）。它允许开发者将对象之间的依赖关系交给Spring容器来管理，从而降低了代码之间的耦合度，提高了代码的可维护性和可扩展性。
- 面向切面编程（AOP）：Spring框架提供了面向切面编程的支持，使得开发者能够轻松实现跨多个对象的交叉功能，如日志记录、安

全性、事务管理等。

- 模块化设计：Spring框架被设计为一系列模块，每个模块都专注于特定的功能，如Spring MVC、Spring Boot、Spring Security等。这些模块可以独立使用或者组合使用，以满足不同的需求。

2. 主要模块

- Spring Core：提供了框架的基本功能，包括控制反转和依赖注入的实现。
- Spring Context：提供了对应用程序上下文的管理，包括JNDI、EJB、电子邮件、国际化、校验和调度等功能。
- Spring AOP：实现了面向切面编程的功能，支持声明式事务管理等。
- Spring DAO：提供了对JDBC的抽象层，简化了数据库访问和错误处理。
- Spring ORM：集成了多个ORM框架，如Hibernate、iBatis等，提供了对象关系映射的支持。
- Spring Web：提供了对Web应用程序的支持，包括Web上下文模块和MVC框架等。

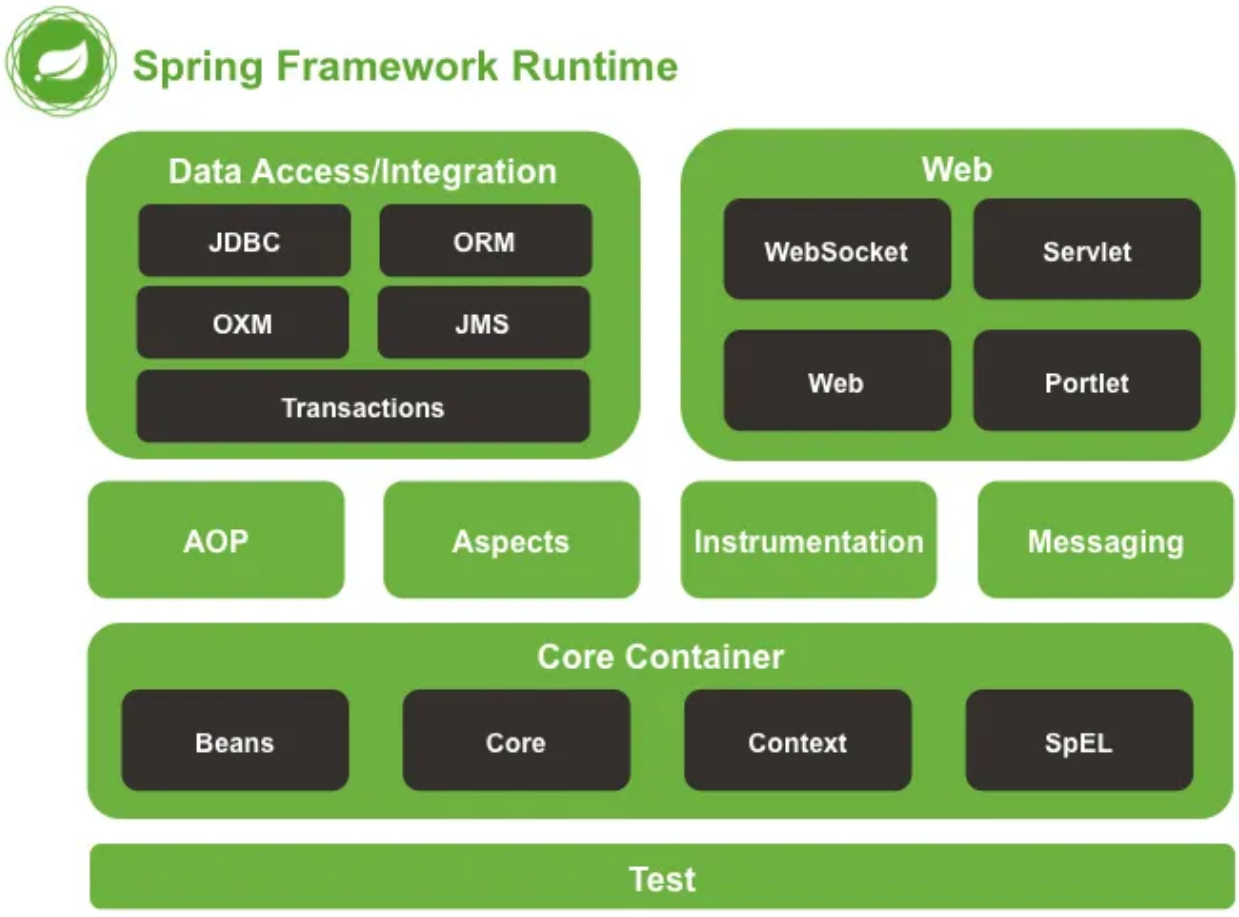
3. 优点

- 简化开发：Spring框架通过提供丰富的功能和工具，简化了Java应用程序的开发过程。
- 降低耦合：通过控制反转和依赖注入等技术，降低了代码之间的耦合度，提高了代码的可维护性和可扩展性。

- 易于测试：Spring框架支持声明式事务管理和各种测试框架，使得开发者能够更容易地对应用程序进行测试。
- 易于集成：Spring框架的设计使得它非常容易与其他组件和框架进行集成，如数据库访问、消息传递、云服务等。

Spring Framework

Spring Framework是Spring生态圈中最基础的项目是其他项目的根基。



Core Container作为Spring框架的核心部分，为Spring应用程序的开发提供了基础的支撑和丰富的功能。通过Core、Beans、Context和

Expression Language这四个模块的协同工作，Spring框架能够实现对象的解耦、依赖注入、资源管理和事件处理等核心功能，为开发者提供了一个强大而灵活的开发平台。

Core模块包含Spring框架基本的核心工具类，是其他一切组件的基本核心。这个模块为Spring提供了底层的支撑，使得其他模块能够在此基础上构建。

Beans模块是所有应用都要用到的，它包含了访问配置文件、创建和管理bean以及进行IoC/DI（控制反转/依赖注入）操作相关的所有类。

Context模块在Core和Beans模块的基础上，Context模块提供了一种类似于JNDI的访问机制，用于访问系统外部的资源。它还继承了Beans模块的特性，并为Spring核心提供了大量的扩展，如国际化、事件传播、资源加载、EJB以及JMX的支持等。

Expression Language（表达式语言）模块是JSP规范中定义的Unified Expression Language（统一表达式语言）的一个扩展，用于在运行时查询和操作对象。它支持设置/获取属性的值、属性的分配、方法的调用、访问数组上下文等操作。

AOP（Aspect-Oriented Programming，面向切面编程）是一种编程范式，它允许开发者将横切关注点（cross-cutting concerns）从业务逻辑中分离出来，从而提高了代码的可维护性、可重用性和模块化。横切关注点指的是那些跨越多个模块或类的公共行为或逻辑，如日志记录、事务管理、安全控制等。

"Aspects"（切面）是AOP（面向切面编程）的一个核心概念。切面用于封装那些跨越多个类和模块的公共行为或逻辑，如日志记录、事务管

理、安全控制等。这些行为通常被称为横切关注点，因为它们不是业务逻辑的一部分，但却对多个业务逻辑有影响。

核心概念

IOC

IOC（Inversion of Control，控制反转）是一种设计原则，用于减少代码间的耦合度。在传统的编程方式中，程序的控制流由程序内部的控制结构（如顺序结构、选择结构、循环结构等）来决定，这通常会导致模块间的紧密耦合，使得代码的维护和扩展变得困难。而IOC通过引入外部容器（如Spring框架中的IoC容器）来管理对象的生命周期和依赖关系，从而将控制权从程序内部转移到外部容器，实现了程序的解耦和模块化。

IOC的核心思想主要有两点：

1. 依赖关系的转移：在传统的编程方式中，对象之间的依赖关系通常是通过在代码中直接创建对象来实现的。这种方式使得对象之间的耦合度很高，一旦某个对象的实现发生变化，就需要修改所有依赖它的对象。而在IOC中，对象的依赖关系不再由代码直接创建，而是由外部容器来提供。这样，当依赖对象发生变化时，只需要修改外部容器的配置，而无需修改代码本身。
2. 控制权的转移：在传统的编程方式中，程序的控制权完全掌握在程序员手中，由程序员来决定何时创建对象、何时销毁对象、何时调用方法等。而在IOC中，控制权被转移到了外部容器手中。外部容器负责对象的生命周期管理（如创建、初始化、销毁等）和依赖关系

的注入，程序员只需要在代码中声明所需的依赖，而无需关心这些依赖是如何被创建和注入的。

Spring框架中的IoC容器是实现IOC思想的关键组件。 同时，IoC容器还会负责将对象的依赖关系注入到对象中，以确保对象能够正常工作。

IOC的好处包括：

- 降低耦合度：通过外部容器来管理对象的依赖关系，减少了代码间的耦合度，使得代码更加模块化和可维护。
- 提高可扩展性：由于对象的依赖关系是通过配置来管理的，因此当需要扩展系统时，只需要修改配置文件或注解信息，而无需修改代码本身。
- 简化代码：在IoC模式下，程序员不再需要编写大量的代码来管理对象的生命周期和依赖关系，这些工作都交给了IoC容器来完成，从而简化了代码的开发和维护工作。

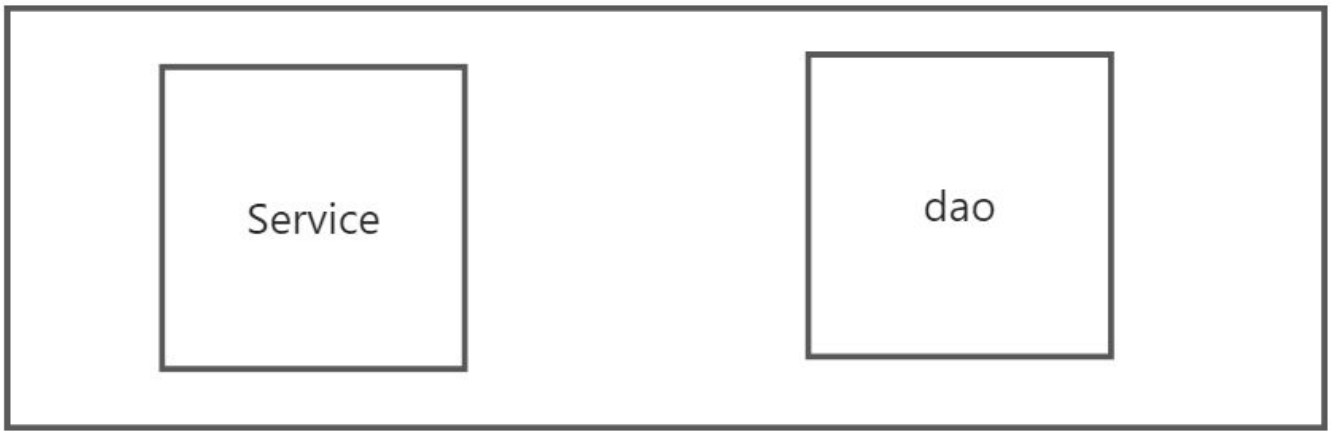
使用对象时，由主动new产生对象转换为由外部提供对象，此过程中对象创建控制权由程序转移到外部，此思想称为控制反转。

Spring提供了一个容器，称为IOC容器，用来充当IOC思想中的“外部”。

Java

```java
public class BookServiceImpl implements BookService{
    private BookDao bookDao;
    public void save(){
        bookDao.save();
    }
}
```

```java
public class BookDaoImpl implements BookDao {
    public void save(){
        System.out.println("book dao save ...");
    }
}
```



Bean

IOC容器负责对象的创建、初始化等工作，被创建或被管理的对象在IOC容器中称为Bean。

DI

DI（Dependency Injection，依赖注入）是实现控制反转（Inversion of Control，IoC）的一种具体方式。在依赖注入中，一个对象（被依赖对象）的依赖项（通常是其他对象）不是由对象本身在内部创建的，而是由外部传入或“注入”到对象中的。这种方式降低了对象之间的耦合度，提高了代码的可维护性和可扩展性。

依赖注入的核心思想是将对象的创建和依赖关系的建立从对象内部转移到外部，从而实现了对象之间的解耦。具体来说，依赖注入可以分为以下几个步骤：

1. 声明依赖：在需要使用其他对象的类中，通过构造函数、setter方法或接口注入等方式声明所需的依赖项。
2. 创建依赖：依赖项（即其他对象）的创建和管理由外部容器（如Spring框架中的IoC容器）负责。容器会根据配置信息或注解信息来创建依赖项，并将它们存储在容器中。
3. 注入依赖：当容器创建了一个对象（即被依赖对象）的实例时，它会根据该对象的依赖声明，从容器中查找相应的依赖项，并将它们注入到对象中。这个过程是自动完成的，无需程序员手动编写代码来实现。

依赖输入的目标：充分解耦。

IOC编码思路

1. 管理什么？（Service和Dao）
2. 如何将被管理的对象告知IOC容器？（配置）
3. 被管理的对象交给IOC容器，如何获取到IOC容器？（接口）
4. IOC容器得到后，如何从容器中获取bean？（接口方法）
5. 使用Spring导入哪些依赖？（pom.xml）

XML

```xml
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>6.1.13</version>
</dependency>
```

在resources下创建applicationContext.xml文件作为配置文件：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
  <!-- 配置bean -->
  <bean id="jobTypeDao" class="org.example.daoimpl.JobTypeDaoImpl"/>

  <bean id="jobTypeService" class="org.example.service.JobTypeService"/>

</beans>
```

获取IOC容器，将配置文件作为参数传入。

```java
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
JobTypeDao jobTypeDao = (JobTypeDao) ctx.getBean("jobTypeDao");
jobTypeDao.save();
```

DI编码思路

1. 基于IOC管理bean
2. Service不再通过new创建Dao对象
3. 如何不使用new创建Dao对象?
4. Service与Dao的关系是什么?

通过property在Service内部配置与Dao的关系。

```xml
<bean id="jobTypeService" class="org.example.service.JobTypeService">
  <property name="jobTypeDao" ref="jobTypeDao" />
</bean>
```

其中name是绑定的Dao，ref是在Service中使用的属性。

bean基础配置

基础配置

- id：定义bean的唯一标识
- class：定义bean的类名

别名配置

- name：给bean添加别名，多个别名使用空格或者逗号隔开

```xml
<bean id="jobTypeDao" name="jobTypeDao2 jobTypeDao3" class="org.example.daoimpl.JobTypeDaoImpl"/>
```

使用别名之后，DI可以使用别名进行注入。

作用范围配置

Spring默认创建的对象是单例模式。

```java
JobTypeService jobTypeService1 = (JobTypeService) ctx.getBean("jobTypeService");
JobTypeService jobTypeService2 = (JobTypeService) ctx.getBean("jobTypeService");
System.out.println(jobTypeService1);
System.out.println(jobTypeService2);
```

- scope：设置模式

```java
<bean id="jobTypeDao" scope="prototype" class="org.example.daoimpl.JobTypeDaoImpl"/>
```

为什么默认是单例？什么场景适合单例？

bean实例化

bean本质上就是对象，创建bean使用构造方法完成。

通过给Service或者Dao的类添加构造方法判断是否调用构造方法。

默认调用无参构造方法。

静态工厂

通过给bean添加factory-method属性配置工厂方法，注意方法必须是public。

```xml
<bean id="jobTypeService" scope="prototype" class="org.example.service.JobTypeService" factory-method="JobTypeServiceFactory">
  <property name="jobTypeDao" ref="jobTypeDao" />
</bean>
```

实例工厂的静态方法

```xml
<bean id="jobTypeDaoFactory" class="org.example.factory.JobTypeDaoFacotry"/>

<bean id="jobTypeDao" factory-bean="jobTypeDaoFactory" factory-method="createJobTypeDao" />
```

工厂bean

```java
package org.example.factory;

import org.example.dao.JobTypeDao;
import org.example.daoimpl.JobTypeDaoImpl;
import org.example.service.JobTypeService;
import org.springframework.beans.factory.FactoryBean;

public class JobTypeDaoFactoryBean implements FactoryBean<JobTypeDao> {
    @Override
    public JobTypeDao getObject() throws Exception {
        return new JobTypeDaoImpl();
    }

    @Override
    public Class<?> getObjectType() {
        return JobTypeDao.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

实际创建的是getObject中返回的对象。

bean生命周期

从创建到消亡的完整过程。

在Dao中添加init和destroy方法，并在实现类中实现。

```java
package org.example.dao;

public interface JobTypeDao {
    public void save();

    public void init();

    public void destroy();
}
```

在bean中使用init-method和destroy-method添加生命周期事件。

```xml
<bean id="jobTypeDao" name="jobTypeDao2 jobTypeDao3" class="org.example.daoimpl.JobTypeDaoImpl" init-method="init" destroy-method="destroy"/>
```

通过ctx.close()或者ctx.registerShutdownHook();方法关闭bean。

依赖注入

向类中传递数据的方式

1. 普通方法（set方法）
2. 构造方法

DI（依赖注入）在容器中建立了bean直接的依赖关系，如果bean运行需要基本类型如何解决?

1. setter注入
2. 构造器注入

setter注入引用类型

在bean中定义引用类型的属性，并提供可访问的set方法。

```Java
private JobTypeDao jobTypeDao;
public void setJobTypeDao(JobTypeDaoImpl jobTypeDao) {
    this.jobTypeDao = jobTypeDao;
}
```

在property标签的ref属性注入引用类型对象。

```XML
<bean id="jobTypeDao" name="jobTypeDao2 jobTypeDao3" class="org.example.daoimpl.JobTypeDaoImpl" />
<bean id="jobTypeService" scope="prototype" class="org.example.serviceimpl.JobTypeServiceImpl" init-method="init" destroy-method="destroy">
  <property name="jobTypeDao" ref="jobTypeDao"/>
</bean>
```

如果在bean中引入多个其他bean，在property中定义多个依赖即可。

setter注入基本类型

在类中添加基本类型的数据。

```Java
public void setId(int id) {
    this.id = id;
}

private int id;
```

在配置中使用value进行赋值。

```xml
<bean id="jobService" class="org.example.serviceimpl.JobServiceImpl">
  <property name="id" value="1"/>
  <property name="jobDao" ref="jobDao"/>
</bean>
```

构造器注入

```java
public class QuestionServiceImpl implements QuestionService {
    public QuestionServiceImpl(QuestionDao questionDao, int id) {
        this.questionDao = questionDao;
        this.id = id;
    }

    private QuestionDao questionDao;
    private int id;


    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public QuestionDao getQuestionDao() {
        return questionDao;
    }

    public void setQuestionDao(QuestionDao questionDao) {
        this.questionDao = questionDao;
    }

    @Override
    public void save() {

    }
}
```

```xml
<bean id="questionDao" class="org.example.daoimpl.QuestionDaoImpl" />
<bean id="questionService" class="org.example.serviceimpl.QuestionServiceImpl">
  <constructor-arg name="questionDao" ref="questionDao" />
  <constructor-arg name="id" value="1" />
</bean>
```

注意构造器配置参数的name必须是构造器的参数名，且参数顺序要与形参顺序一致。

也可以使用类型代替name实现解耦。

```xml
<bean id="questionDao" class="org.example.daoimpl.QuestionDaoImpl" />
<bean id="questionService" class="org.example.serviceimpl.QuestionServiceImpl">
  <constructor-arg type="org.example.dao.QuestionDao" ref="questionDao" />
  <constructor-arg type="java.util.String" value="1" />
</bean>
```

甚至可以直接通过位置设定。

```xml
<bean id="questionDao" class="org.example.daoimpl.QuestionDaoImpl" />
<bean id="questionService" class="org.example.serviceimpl.QuestionServiceImpl">
  <constructor-arg index="0" ref="questionDao" />
  <constructor-arg index="1" value="1" />
</bean>
```

如何选择DI?

1. 强依赖使用构造器注入，使用setter可能出现对象为null
2. 可选依赖使用setter，更加灵活
3. 优先使用构造器

自动装配依赖

IOC容器根据bean所依赖的资源在容器中自动查找并注入到bean中的过程称为自动装配。

方式：

- 按类型
- 按名称
- 按构造方法

```xml
<bean id="areaService" class="org.example.serviceimpl.AreaServiceImpl" autowire="byType"/>
```

注意当出现相同类型的bean时，按类型装配将无法实现，按名称可以实现，bean中的依赖的setter必须和某一个id相同。

```xml
<bean id="areaService" class="org.example.serviceimpl.AreaServiceImpl" autowire="byName"/>
```

集合注入

- List
- Set
- Map

Java

```java
package org.example.daoimpl;

import org.example.dao.AreaDao;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class AreaDaoImpl implements AreaDao {
    private int[] array;
    private List<String> list;
    private Set<String> set;
    private Map<String, String> map;
    private Properties properties;
    @Override
    public void getArea() {

    }

    public int[] getArray() {
        return array;
    }

    public void setArray(int[] array) {
        this.array = array;
    }

    public List<String> getList() {
        return list;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public Set<String> getSet() {
        return set;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }

    public Map<String, String> getMap() {
```

```
46          return map;
47      }
48
49      public void setMap(Map<String, String> map) {
50          this.map = map;
51      }
52
53      public Properties getProperties() {
54          return properties;
55      }
56
57      public void setProperties(Properties properties) {
58          this.properties = properties;
59      }
60
61
62  }
63
```

配置文件

```xml
<bean id="areaDao" class="org.example.daoimpl.AreaDaoImpl">
        <property name="array">
            <array>
                <value>1</value>
                <value>2</value>
                <value>3</value>
                <value>4</value>
            </array>
        </property>
        <property name="list">
            <list>
                <value>a</value>
                <value>b</value>
                <value>c</value>
                <value>d</value>
            </list>
        </property>
        <property name="set">
            <set>
                <value>a</value>
                <value>a</value>
                <value>c</value>
                <value>b</value>
            </set>
        </property>
        <property name="map">
            <map>
                <entry key="a" value="a" />
                <entry key="b" value="b" />
                <entry key="c" value="c" />
            </map>
        </property>
        <property name="properties">
            <props>
                <prop key="a">a</prop>
                <prop key="b">b</prop>
            </props>
        </property>
    </bean>
```

如果是引用类型，通过ref标签实现：

```xml
<ref bean="beanId" />
```

容器

1. 加载类路径下的配置文件

```java
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
```

2. 从文件系统下加载配置文件

```java
ApplicationContext ctx= new FileSystemXmlApplicationContext("路径");
```

通过反射获取类型：

```java
JobTypeService jobTypeService = ctx.getBean("jobTypeService", JobTypeServic.class);
```

通过类型获取bean（注意bean必须唯一）：

```java
JobTypeService jobTypeService = ctx.getBean(JobTypeServic.class);
```

注解开发

注解开发定义bean

将QuestionDao变成注解开发。

```xml
<bean id="questionDao" class="org.example.daoimpl.QuestionDaoImpl" />
```

给QuestionDaoImp添加@Component注解：

```java
@Component("questionDao")
public class QuestionDaoImpl implements QuestionDao {

    @Override
    public void save() {
        System.out.println("QuestionDaoImpl save");
    }
}
```

修改配置文件，使用context命名空间扫描组件，spring会自动在给定的目录中查找：

```xml
<context:component-scan base-package="org.example.daoimpl"/>
```

注意，修改beans的配置信息，创建context命名空间：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd"
>
</beans>
```

Spring提供@Component的衍生注解：

1. @Controller：用于表现层bean定义
2. @Service：用于业务层bean定义
3. @Repository：用于数据层bean定义

纯注解开发

纯注解开发开启Spring快速化开发。

通过配置文件可以代替applicationContext.xml文件。

创建config目录作为配置包目录，创建SpringConfig.java作为配置类：

```java
@Configuration
@ComponentScan("org.example.daoimpl")
public class SpringConfig {

}
```

配置文件结构被@Configuration代替，扫描配置被@ComponentScan代替。

- @Configuration用于设定类为配置类
- @ComponentScan用于设定扫描路径，只能添加一次，如果有多个数据需要使用数组的形式

在使用时，将ClassPathXmlApplicationContext的实例修改为AnnotationConfigApplicationContext，注意添加配置类为参数。

```java
ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);
QuestionDao service = (QuestionDao) ctx.getBean(QuestionDao.class);
System.out.println(service);
```

bean管理

通过@Scope("singleton")注解给实现类配置单例模式。

```java
@Scope("singleton")
```

通过@PostConstruct注解配置初始生命周期，通过@PreDestroy配置销毁生命周期。

```java
@PostConstruct
public void init() {
    System.out.println("QuestionDaoImpl init");
}

@PreDestroy
public void destroy() {
    System.out.println("QuestionDaoImpl destroy");
}
```

注解开发依赖注入

通过@Autowired注解给字段添加注入，根据类型查找bean，无须再通过setter方法。

```java
@Autowired
private QuestionDao questionDao;
```

默认按照类型装配，如果要按照名称装配，使用@Qualifier注解可以根据名称查找类型：

```java
@Autowired
@Qualifier("questionDaoImpl")
private QuestionDao questionDao;
```

通过@Value注解给基本类型添加注入。

```java
@Value("100")
private int id;
```

如果需要从外部导入数据，通过@PropertySource给SpringConfig类添加数据源。

```java
@Configuration
@ComponentScan({"org.example.dao", "org.example.daoimpl", "org.example.service","org.example.serviceimpl"})
@PropertySource("jdbc.properties")
public class SpringConfig {

}
```

修改@Value参数：

```java
@Value("${host}")
private int host;
```

使用第三方bean

在SpringConfig类中添加返回DataSource方法，创建第三方类实例，并返回。

```java
@Bean
public DataSource datasource(){
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://1ocalhost:3306/tencent_careers");
    ds.setUsername("root");
    ds.setPassword("123456");
    return ds;
}
```

Spring整合Mybatis

要和 Spring 一起使用 MyBatis，需要在 Spring 应用上下文中定义至少两样东西：一个 SqlSessionFactory 和至少一个数据映射器类。

```java
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
try (SqlSession session = sqlSessionFactory.openSession()) {
    List<Admin> adminList = session.selectList("AdminMapper.selectAllAdmin");
    System.out.println(adminList);
    req.setAttribute("adminList", adminList);
}
```

安装坐标

```xml
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>3.0.4</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.2.23</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>6.1.13</version>
</dependency>
```

获取SqlSessionFactory

```java
@Bean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
    SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
    factoryBean.setDataSource(dataSource);
    return factoryBean.getObject();
}
```

SqlSessionFactory 需要一个 DataSource（数据源）。这可以是任意的 DataSource，只需要和配置其它 Spring 数据库连接一样配置它就可以了。

获取SQLSession

在 MyBatis 中使用 SqlSessionFactory 来创建 SqlSession。 一旦获得一个 session 之后，可以使用它来执行映射了的语句，提交或回滚连接，最后，当不再需要它的时候，可以关闭 session。 使用 MyBatis-Spring 之后不再需要直接使用 SqlSessionFactory 了，因为 bean 可以被注入一个线程安全的 SqlSession，它能基于 Spring 的事务配置来自动提交、回滚、关闭 session。

SqlSessionTemplate 是 MyBatis-Spring 的核心。作为 SqlSession 的一个实现，这意味着可以使用它无缝代替你代码中已经在使用的 SqlSession。 SqlSessionTemplate 是线程安全的，可以被多个 DAO 或映射器所共享使用。

```java
@Bean
public SqlSessionTemplate sqlSession(DataSource dataSource) throws Exception {
    return new SqlSessionTemplate(sqlSessionFactory(dataSource));
}
```

编写数据映射

```java
@Mapper
public interface AdminMapper {
    @Select("SELECT * FROM tb_admin WHERE id = 1")
    Admin getAdmin(@Param("id") String id);
}
```

注入映射类

```java
@Bean
public MapperFactoryBean<AdminMapper> adminMapper(DataSource dataSourceConfig) throws Exception {
    MapperFactoryBean<AdminMapper> factoryBean = new MapperFactoryBean<>(AdminMapper.class);
    factoryBean.setSqlSessionFactory(sqlSessionFactory(dataSourceConfig));
    return factoryBean;
}
```

使用Mybatis

直接获取bean

```java
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);
AdminMapper adminMapper = (AdminMapper) context.getBean("adminMapper");
```

在其他类中使用

```java
@Autowired
private AdminMapper adminMapper;
```

扫描映射

```java
@MapperScan("com.example.mapper")
```

AOP

面向切面编程，指导开发者如何组织程序结构。

AOP可以在不修改原有代码的情况下，对方法进行权限控制、性能监控等操作。

核心概念

切面（Aspect）

切面是横切关注点被模块化的特殊对象，通常是一个类。它包含了横切逻辑的定义，以及连接点的定义。

在Spring AOP中，使用@Aspect注解的类被视为切面。

连接点（JoinPoint）

连接点是程序执行过程中能够插入切面的一个点，如方法的执行、异常的处理等。

在Spring AOP中，连接点通常指的是方法的执行点。

切入点（PointCut）

切入点是一组连接点的总称，用于指定某个增强（Advice）应该在何时被调用。

它通过AspectJ切入点表达式来定义，用于匹配特定的连接点。

通知（Advice）

通知是切面在特定连接点上执行的动作，它是增强代码的逻辑部分。

Spring AOP支持多种类型的通知，如前置通知（Before）、后置通知（After）、返回通知（AfterReturning）、异常通知（AfterThrowing）和环绕通知（Around）。

目标对象（Target）

目标对象是被增强的对象，即包含主业务逻辑的对象。

代理对象（Proxy）

代理对象是Spring AOP通过代理模式创建的对象，它包含了目标对象的所有方法，并在方法调用时加入增强的逻辑。

入门案例

安装依赖

XML

```xml
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.9.22.1</version>
</dependency>
```

编写切面

```java
package org.example.aop;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

@Aspect
@Component
public class Log {
    @Pointcut("execution(* org.example.*.*(..))")
    private void pointcut() {}

    @Before("pointcut()")
    public void logCurrentDatetime() {
        // 获取当前时间
        LocalDateTime now = LocalDateTime.now();
        // 定义时间格式
        DateTimeFormatter formatter = DateTimeFormatter.ofPatter
n("yyyy-MM-dd HH:mm:ss");
        // 格式化当前时间
        String formattedDateTime = now.format(formatter);
        // 输出格式化后的时间
        System.out.println(formattedDateTime);
    }
}
```

其中@Aspect注解声明这是一个切面类，@Component注解使得这个类能够被Spring容器管理，使用@Pointcut注解定义一个切入点表达式，这里匹配org.example包及其子包中所有类的所有方法，使用@Before注解声明这是一个前置通知，它会在切入点匹配的方法执行之前执行，同理还有@After注解。

启用切面

在Spring配置类中，添加Bean扫描，然后通过@EnableAspectJAutoProxy启用切面。

```java
@Configuration
@ComponentScan({"org.example.dao", "org.example.daoimpl", "org.example.service","org.example.serviceimpl", "org.example.aop"})
@PropertySource("jdbc.properties")
@EnableAspectJAutoProxy
public class SpringConfig {

}
```

使用切面

直接在切面所定义的切入点指定的位置创建实例并调用方法即可。

注意点

1. 添加@EnableAspectJAutoProxy注解以确保启用切面
2. 切面类也是一个类，Spring如果要启用切面必须获取实例，在ComponentScan的参数中，注意添加切面所在的路径供Spring扫描
3. Spring AOP的代理默认是基于JDK动态代理（针对接口）或CGLIB代理（针对类）来实现的，Spring会创建一个JDK动态代理对象来包装目标对象。在这种情况下，代理对象将实现与目标对象相同的接口，并拦截对这些接口方法的调用。如果目标对象没有通过接口来引用（即它是直接通过类来引用的），那么Spring会尝试使用CGLIB来创建一个代理类，这个代理类是目标类的子类，并会拦截对子类（也就是目标类）中所有非final方法的调用。

问题排查

如果切面没有按预期工作，可能是由以下原因造成的：

- 切点表达式不正确：确保你的切点表达式正确地指向了你想要拦截的方法。
- 标对象是一个Spring Bean，并且Spring能够创建其代理对象。
- 代目理配置问题：如果你在使用自定义的代理配置（比如通过@EnableAspectJAutoProxy的proxyTargetClass属性），请确保你的配置是正确的。
- 方法访问权限：如果你的方法是非public的（比如protected或private），那么CGLIB代理可能无法拦截这些方法的调用（JDK动态代理只能拦截接口中的public方法）。
- final方法：如果方法是final的，那么它不能被CGLIB代理所拦截。
- 静态方法：静态方法不能被Spring AOP拦截，因为它们不属于类的实例，而是属于类本身。

```java
@Configuration
@ComponentScan({"org.example.dao", "org.example.daoimpl", "org.example.service","org.example.serviceimpl", "org.example.aop"})
@PropertySource("jdbc.properties")
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class SpringConfig {

}
```

实战案例

编写一个用于处理请求的切面类，连接点是所有Servlet的doGet和doPost请求，有两个通知，分别是前置通知，用于验证登录凭证以及输

出请求方法，请求url以及请求参数，后置通知用于输出请求响应时的时间。

注意：切入点指的是哪些连接点会被增强处理，其并不是一个具体的内容，需要通过切入点表达式进行声明定义方可使用。

工作流程

1. 启动Spring容器
2. 读取所有切面配置及切入点
3. 初始化bean，验证bean对应的类是否匹配到切入点
    a. 匹配成功，创建目标对象的代理对象
    b. 匹配失败，直接创建对象
4. 获取bean执行方法
    a. 匹配成功时，根据代理对象的运行模式完成操作
    b. 匹配失败时，直接调用方法完成操作

切入点表达式

切入点表达式用于定义哪些连接点（如方法调用）应该被切面（Aspect）的增强逻辑所拦截。

```Java
execution([访问控制权限修饰符] 返回值类型 [全限定类名]方法名(形式参数列表) [异常])
```

1. execution()：这是切入点表达式的主体，表示这是一个方法执行连接点的表达式。
2. 访问控制权限修饰符（可选）：如public、protected、private等，

表示匹配具有特定访问权限的方法。如果省略，则默认匹配所有访问权限的方法。

3. 返回值类型：表示匹配方法的返回类型。*表示匹配任意返回类型。
4. 全限定类名（可选）：表示匹配特定类或接口的方法。可以使用*通配符来匹配任意类名或包名。例如，com.example.service.*表示com.example.service包下的所有类；com.example..*表示com.example包及其所有子包下的所有类。
5. 方法名：表示匹配的方法名称。可以使用*通配符来匹配任意方法名，或者使用特定模式（如*To匹配所有以To结尾的方法）。
6. 形式参数列表（可选）：表示匹配方法的参数类型和数量。()表示没有参数的方法；(..)表示任意数量和类型的参数；(String, int)表示第一个参数是String类型，第二个参数是int类型的方法。
7. 异常（可选）：表示匹配抛出特定异常的方法。如果省略，则默认匹配不抛出异常或抛出任意异常的方法。

通知类型

通知类型定义了切面中的增强逻辑应该在何时以及如何应用到目标连接点上。

前置通知（Before Advice）

- 在目标方法执行之前执行。
- 它不能阻止目标方法的执行流程（除非它抛出一个异常）。
- 使用@Before注解进行声明。

后置通知（After Advice）：

- 在目标方法执行之后执行，无论目标方法是否抛出异常。
- 它通常用于执行一些清理工作，如释放资源等。
- 使用 `@After` 注解进行声明。需要注意的是，这种通知类型在Spring AOP中通常被称为“最终通知（Finally Advice）”，但在其他上下文中可能直接称为“后置通知”。

返回后通知（After Returning Advice）：

- 在目标方法正常执行完毕后执行。
- 如果目标方法抛出异常，则不会执行此通知。
- 使用 `@AfterReturning` 注解进行声明，并可以通过 `returning` 属性访问目标方法的返回值。

抛出异常后通知（After Throwing Advice）：

- 在目标方法抛出异常后执行。
- 它允许你对异常进行处理，如记录日志、抛出新的异常等。
- 使用 `@AfterThrowing` 注解进行声明，并可以通过 `throwing` 属性访问抛出的异常对象。

环绕通知（Around Advice）：

- 环绕目标方法执行，即在目标方法执行前后都可以执行自定义的行为。
- 它是最强大的通知类型，因为它可以控制目标方法的执行流程，包括是否执行目标方法、修改目标方法的返回值或抛出异常来中断执行。
- 使用 `@Around` 注解进行声明，并通过 `ProceedingJoinPoint` 对象来调用目标

方法。

实战

通过aop通知获取数据。